**Department of Computer Science**

# INCREMENTAL RUNTIME VERIFICATION OF PROBABILISTIC SYSTEMS

Vojtěch Forejt
Marta Kwiatkowska
David Parker
Hongyang Qu
Mateusz Ujma

RR-12-05

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD

# Incremental Runtime Verification of Probabilistic Systems (Extended Version)

Vojtěch Forejt[1], Marta Kwiatkowska[1], David Parker[2], Hongyang Qu[1], and Mateusz Ujma[1]

[1] Department of Computer Science, University of Oxford, Oxford, UK
[2] School of Computer Science, University of Birmingham, Birmingham, UK

**Abstract.** Probabilistic verification techniques have been proposed for runtime analysis of adaptive software systems, with the verification results being used to steer the system so that it satisfies certain Quality-of-Service requirements. Since systems evolve over time, and verification results are required promptly, efficiency is an essential issue. To address this, we present incremental verification techniques, which exploit the results of previous analyses. We target systems modelled as Markov decision processes, developing incremental methods for constructing models from high-level system descriptions and for numerical solution using policy iteration based on strongly connected components. A prototype implementation, based on the PRISM model checker, demonstrates performance improvements on a range of case studies.

## 1 Introduction

Computerised systems are prevalent in our daily life and a growing number of our everyday activities depend on their correctness. Many of these systems exhibit probabilistic behaviour: physical devices may fail, communication media are lossy and protocols use randomisation. Hence, traditional verification techniques do not suffice to ensure their correct behaviour. *Probabilistic model checking* is an automated method of verifying quantitative properties of these systems. An example of such a property is: "the web service successfully delivers a response within 5ms with probability at least 0.99".

It has recently been proposed to use probabilistic model checking for *runtime* verification of *adaptive* systems [2], where quantitative verification is used to steer a system such that it satisfies formally specified Quality-of-Service (QoS) requirements. The framework of [2] is illustrated in Figure 1. It comprises a computer system exhibiting probabilistic behaviour, a monitoring module that observes its behaviour and a reconfiguration component, which issues it instructions. Requirements to be fulfilled are verified against a high-level model of system's behaviour, which is parametrised using data from the monitoring module. The results of verification are then forwarded to the reconfiguration module,

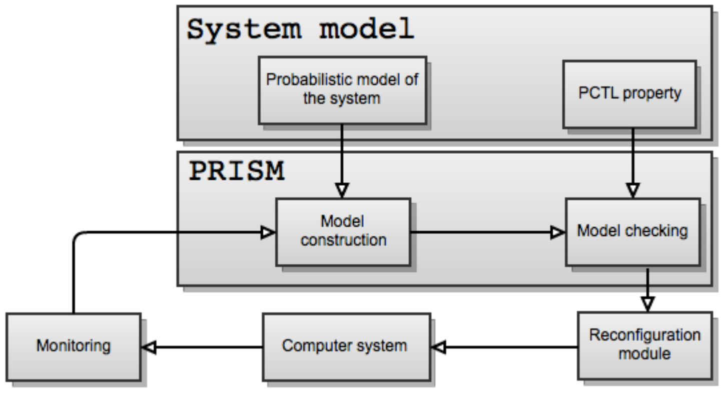

**Fig. 1.** Runtime verification of probabilistic systems using PRISM [2].

which directs the system accordingly. In this paper, we use the probabilistic model checker PRISM [10] for analysing system models.

A real-world example of a computer system that could be monitored with such techniques is a dynamically changing network in which joining devices establish local IP addresses using the ZeroConf protocol. This protocol is based on the random selection of an IP address, followed by the transmission of several probes: messages that enquire about the availability of the chosen address. To improve the network QoS, we would want to minimise the probability of choosing a conflicting IP address. Parameters that influence this probability include the number of hosts in the network and the number of probes sent before claiming a given IP address. Such parameters could be monitored and used by a probabilistic model checker to compute the probability of a conflict. The results would then be forwarded to the reconfiguration module which can, for example, increase the number of probes sent, if the probability of the conflict is too high.

In this paper, our aim is to optimise the performance of runtime verification for probabilistic systems. Since the systems being verified change dynamically and the results of verification are needed promptly to steer the system, efficiency is essential. We consider *incremental* verification techniques, which exploit the results of previous analyses following a small change to the system being verified.

We target systems modelled as Markov decision processes (MDPs), a widely used model for systems exhibiting both *probabilistic* behaviour (e.g., using a randomly generated IP address) and *nondeterministic* behaviour (e.g., due to concurrency between network devices). We present incremental techniques for the two main phases of probabilistic verification: model construction, which exhaustively constructs an MDP from a high-level model description, and quantitative verification, which applies numerical techniques to determine the correctness of a system requirement, formally specified in temporal logic. For the former, we propose a technique that infers all states that have to be visited in the incremental step. For the later, we use policy iteration, optimised using a decomposition of the system into strongly connected components, and performed incrementally by re-using policies between verification runs. We have implemented our techniques in a prototype extension of PRISM and illustrate the benefits of our approach on a set of benchmark models.

### 1.1 Related Work

Various techniques have been developed that use model checking at runtime; see [1] for a discussion and further references. There is also increasing interest in incremental model checking techniques. Of particular relevance here are those for probabilistic systems. Wongpiromsarn et al. [19] studied incremental model construction for increasing numbers of system components. In contrast, we focus on changes within a fixed set of components. Filieri et al. [7] presented efficient incremental verification for the simpler model of discrete-time Markov chains using parametric techniques, but their method is subject to an exponential blow up when applied to MDPs and does not handle structural model changes. Kwiatkowska et al. [14] proposed incremental methods for MDPs based on a decomposition into strongly connected components. We consider model changes at the modelling language description level, which [14] does not, and also permit changes in model structure, rather than just transition probabilities. Other topics related to the work presented in this paper include system reconfiguration [4,16] and monitoring [5,20], which we do not consider.

## 2 Preliminaries

In this paper, we use $\mathbb{N}$, $\mathbb{R}$ and $\mathbb{Q}$ for the sets of natural, real and rational numbers, respectively. A *probability distribution* over a finite or countable set $A$ is a function $d : A \to [0,1]$ such that $\sum_{a \in A} d(a) = 1$. By $\mathit{Dist}(A)$ we denote the set of all probability distributions over $A$.

### 2.1 Markov Decision Processes

Markov decision processes (MDPs) are a commonly used formalism when there is a need to capture both stochastic and nondeterministic behaviour. Formally, an MDP is a tuple $\mathcal{M}{=}(S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ where $S$ is a finite set of *states*, $\overline{s} \in S$ is an initial state, $\alpha_{\mathcal{M}}$ is a finite *action* alphabet, $\delta_{\mathcal{M}} : S{\times}\alpha_{\mathcal{M}} \to \mathit{Dist}(S)$ is a (partial) probabilistic *transition function* and $L : S \to 2^{AP}$ is a *labelling function* mapping states to sets of atomic propositions from a set $AP$.

A transition from a state $s \in S$ is performed by taking an *action* $a \in \alpha_{\mathcal{M}}$, and then choosing a successor $s'$ according to the probability distribution $\delta_{\mathcal{M}}(s, a)$. We use $A_{\mathcal{M}}(s)$ to denote the set of all $a$ such that $\delta_{\mathcal{M}}(s, a)$ is defined. A finite or infinite *path* is a sequence $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots$ (or just $s_0 a_0 s_1 a_1 s_2 \ldots$) where $s_i \in S$, $a_i \in A_{\mathcal{M}}(s_i)$ and $\delta_{\mathcal{M}}(s_i, a_i)(s_{i+1}) > 0$ for all $i$. For a finite path $\rho$, we use $\mathit{last}(\rho)$ to denote its last state. The sets of all finite and infinite paths are denoted $\mathit{FPath}_{\mathcal{M}}$ and $\mathit{IPath}_{\mathcal{M}}$, respectively, and the set of all such paths beginning in a state $s$ are denoted $\mathit{FPath}_{\mathcal{M},s}$ and $\mathit{IPath}_{\mathcal{M},s}$, respectively.

A *strongly connected component* (SCC) in the MDP is a set $C \subseteq S$ of states that is strongly connected (there is a path between any two states in $C$) and maximal (no superset of $C$ is also strongly connected).

**Adversaries.** The choice of actions in an MDP is nondeterministic, and is resolved using an *adversary* (also known as a policy or strategy), which is a function $\sigma : \mathit{FPath}_{\mathcal{M}} \rightarrow \mathit{Dist}(\alpha_{\mathcal{M}})$ satisfying $\sigma(\rho)(a){>}0$ only if $a \in A_{\mathcal{M}}(\mathit{last}(\rho))$. The set of all adversaries of $\mathcal{M}$ is $\mathit{Adv}_{\mathcal{M}}$. A class of adversaries of particular interest are *memoryless deterministic* adversaries, which choose the distribution based on the last state of the path only, and always assign a Dirac distribution (i.e. choose a single action with probability 1).

An MDP $\mathcal{M}$, together with a state $s$ and an adversary $\sigma \in \mathit{Adv}_{\mathcal{M}}$, induce a probability space $\mathit{Pr}^{\sigma}_{s}$ on $\mathit{IPath}_{\mathcal{M},s}$, defined in a standard way [9]. For a measurable set $\mathit{\Pi} \subseteq \mathit{IPath}_{\mathcal{M},s}$, we can then study the minimum or maximum probabilities of $\mathit{\Pi}$, over all adversaries:

$$\mathit{Pr}^{\min}_{s}(\mathit{\Pi}) \stackrel{\text{def}}{=} \inf_{\sigma \in \mathit{Adv}_{\mathcal{M}}} \mathit{Pr}^{\sigma}_{s}(\mathit{\Pi}) \quad \text{and} \quad \mathit{Pr}^{\max}_{s}(\mathit{\Pi}) \stackrel{\text{def}}{=} \sup_{\sigma \in \mathit{Adv}_{\mathcal{M}}} \mathit{Pr}^{\sigma}_{s}(\mathit{\Pi}).$$

## 2.2 The PRISM Modelling Language

In this paper, we use the probabilistic model checker PRISM, which supports the analysis of several types of probabilistic models, including MDPs. Models to be analysed in PRISM are specified in the PRISM modelling language, a textual formalism based on guarded commands. We now briefly describe the syntax and semantics of this language. We only concentrate on the part which is most relevant for this paper; for a more detailed description, see [17].

A PRISM file consists of finite-domain variables $x_1, \ldots x_n$ and a set of $k$ *modules* (for simplicity, we assume that all variables are global and so every module can access and change any variable). A module consists of a list of guarded commands of the form:

$$[a]\ g \rightarrow \lambda_1 : u_1 + \cdots + \lambda_m : u_m$$

where $a$ is an action, $g$ is a *guard*, which is a boolean expression over the variables, $\lambda_i \in (0, 1]$ for $1 \leq i \leq m$ are positive real numbers summing up to 1, and $u_i$ are *variable updates.* A variable update is a conjunction of operations of the form $x_i' = f(x_1, \ldots x_m)$ which change values of variables, for example $(x_1{'}{=}x_1 + 1)\&(x_2{'}{=}x_3 \cdot x_3)$ is a variable update that increments $x_1$ by one and assigns the square of the value of $x_3$ to $x_2$.

The modules in a PRISM file are combined through *parallel composition.* For simplicity, we define this at the level of the syntax of a PRISM file. The parallel composition $M_1|[A]|M_2$, of modules $M_1$ and $M_2$, synchronising over a set of actions $A$, consists of all commands of $M_1$ and $M_2$ which are not labelled with an action from $A$ and commands of the form:

$$[a]\ g\&g' \to \sum_{i=1}^{n}\sum_{j=1}^{m} \lambda_i \cdot \lambda_j : u_i \& u'_j$$

where $[a]\ g \to \lambda_1 : u_1 + \cdots + \lambda_n : u_n$ and $[a]\ g' \to \lambda'_1 : u'_1 + \cdots + \lambda'_m : u'_m$ are commands of $M_1$ and $M_2$, and $a \in A$.

A PRISM file determines a *system module*, which is the parallel composition $M_1|[A_1]|M_2|[A_2]\cdots[A_{k-1}]|M_k$ of all modules $M_1, \dots M_k$ in the file, where $A_i$ is defined inductively as the intersection of the sets of actions of $M_1|[A_1]|\cdots M_i$ and $M_{i+1}$. The semantics of the PRISM file is then given by defining the MDP that corresponds to the system module $M$. This is an MDP $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ defined as follows. Let $x_1, \dots x_n$ be all variables of $M$ and $V_1, \dots V_n$ their possible values. We put $S = V_1 \times \cdots V_n$, $\overline{s} = (\overline{v}_1, \dots \overline{v}_n)$, where $\overline{v}_i$ is the initial value of the variable $x_i$, and let $\alpha_{\mathcal{M}}$ be equal to the set of all actions of all commands in $M$. The transition function $\delta_{\mathcal{M}}$ is defined in terms of $\mathcal{M}$'s commands. Since we work in this paper with MDPs, we will make the assumption that, for each state $(v_1, \dots, v_n)$ and action $a$, there is at most one $a$-labelled command of $M$ whose guard $g$ is satisfied in $(v_1, \dots, v_n)$, i.e., where $g[v_1/x_1, \dots v_n/x_n]$ evaluates to true. Then, $\delta_{\mathcal{M}}((v_1, \dots v_n), a)$ is defined for a state $(v_1, \dots, v_n)$ if and only if such a command exists. When it does, let this (unique) command be:

$$c \quad \equiv \quad [a]\ g \to \lambda_1 : u_1 + \cdots + \lambda_m : u_m$$

We define $\delta_{\mathcal{M}}((v_1, \dots v_n), a)$ to be equal to the distribution $d$ which, for each state $(v'_1, \dots v'_n)$, assigns probability equal to the sum of values $\lambda_i$ for $1 \leq i \leq m$ such that applying update $u_i$ to variable values $(v_1, \dots v_n)$ gives $(v'_1, \dots v'_n)$.

Lastly, the labelling function $L$ is also defined in the PRISM file as a set of atomic propositions, each with a corresponding predicate over the variables $x_1, \dots, x_n$, that defines the set of states that satisfy it.

### 2.3 Computing Reachability Probabilities

In this paper, we aim at computing the minimum and maximum probability of reaching a given set of states. This problem forms the basis for verifying various commonly used temporal logics, such as PCTL and LTL, against MDPs. Formally, for a set $T \subseteq S$, we define $\Diamond T = \{\pi \subseteq \mathit{IPath}_{\mathcal{M},s} \mid \pi \text{ contains a state of } T\}$, and compute the values $Pr_s^{\min}(\Diamond T)$ and $Pr_s^{\max}(\Diamond T)$.

Common methods for computing these probabilities are *value iteration*, which is an approximate iterative numerical solution method, and *policy iteration*, which analyses a sequence of adversaries with increasing/decreasing probabilities. We briefly explain how these techniques are used for computing $Pr_s^{\max}(\Diamond T)$; the case $Pr_s^{\min}(\Diamond T)$ is similar.

**Value Iteration.** The value iteration technique relies on the fact that $Pr_s^{\max}(\Diamond T) = \lim_{n\to\infty} x_s^n$ where $x_s^n$ is equal to 1 if $s \in T$, to 0 if $n = 0$, and to:

$$\max_{a\in A} \sum_{s'\in S} \delta_{\mathcal{M}}(s,a)(s') \cdot x_s^{n-1}$$

otherwise. In order to get a sufficient approximation, the numbers $x_s^n$ can be computed for suitably large $n$.

**Policy Iteration.** In policy iteration, we start with an arbitrary memoryless deterministic adversary $\sigma$ and compute the probabilities $Pr_s^\sigma(\Diamond T)$. Then, we determine whether $\sigma$ is optimal by checking that:

$$Pr_s^\sigma(\Diamond T) \geq \max_{a\in A} \sum_{s'\in S} \delta_{\mathcal{M}}(s,a)(s') \cdot Pr_{s'}^\sigma(\Diamond T)$$

for all $s \in S \setminus T$. If $\sigma$ is not optimal, then a new memoryless deterministic adversary is chosen by picking $\arg\max_{a\in A} \sum_{s'\in S} \delta_{\mathcal{M}}(s,a)(s') \cdot Pr_{s'}^\sigma(\Diamond T)$ in each $s \in S$, and policy iteration continues from the beginning with the new adversary. If $\sigma$ is found to be optimal, then the equality $Pr_s^\sigma(\Diamond T) = Pr_s^{\max}(\Diamond T)$ holds.

Again, in practice, we often do not compute the values $Pr_s^\sigma(\Diamond T)$ precisely, but approximate them using a variant of value iteration for a fixed adversary.

## 3 Incremental Model Construction

The first phase of verifying a probabilistic system typically involves *construction* of the probabilistic model to be analysed, i.e. an exhaustive exploration of its state space, based on a high-level model description. In many cases, model construction plays a significant role in the overall performance of a model checker, so it is important to consider techniques for improving model construction time.

In this paper, we focus on models described in the PRISM modelling language. The costly part of model construction is the evaluation of commands and subsequent creation of new states in the MDP being built. We outline a novel method for incremental model construction that is designed to operate after relatively small runtime changes to the structure of the MDP. At the level of the high-level model description, we assume that these changes are made by altering *parameters* of the PRISM model. These are constants from the model description whose value is not determined until runtime. We only consider changes in parameters that occur in guards of commands, which is a common scenario in practice. For simplicity, we do not consider parameters that affect transition probabilities values. Such changes could be handled using the techniques described in [14].

The proposed technique is implemented as an extension of the model construction phase of PRISM's explicit-state model checking engine. This builds an MDP from a PRISM model, based a systematic exploration of its state-space from the initial state using a variant of depth-first-search (DFS). Whenever a new state is discovered, all commands are evaluated for the given state. If a guard of the command is satisfied, then all updates are performed to find the state's successors. The basic idea of our incremental method is to infer the subset of states needing to be rebuilt, reducing the number of commands to be re-evaluated.

We illustrate our approach using an example: a PRISM model [11] of the previously mentioned Zeroconf protocol. The PRISM model description is presented in Figure 2. Parameters of the model are given as *undefined constants*. The constant $N$ represents the number of the devices in the network and the constant $K$ represents the number of probes. We describe the case for changes to $K$ only, but our algorithm supports an arbitrary number of parameters.

```
1 :  mdp
2 :
3 :  // CONSTANTS
4 :  const int N; // number of abstract hosts
5 :  const int K; // number of probes to send
6 :  const double loss; // probability of message loss
7 :
8 :  // PROBABILITIES
9 :  const double old = N/65024; // probability pick an ip address being used
10 : const double new = (1 − old); // probability pick a new ip address
11 :
12 :  ...
13 :
14 : module environment
15 :
16 :  ...
17 :
18 : endmodule
19 :
20 : module host0
21 :
22 :      probes  : [0..K];
23 :
24 :  ...
25 :
26 :      // send probe
27 :      [send] l=2 & x=2  & probes<K  →  (x′=0) & (probes′=probes + 1);
28 :      // sent K probes and waited 2 seconds
29 :      [] l=2 & x=2 & probes=K  →  (l′=3) & (probes′=0) & (coll′=0) & (x′=0);
30 :
31 :  ...
32 :
33 : endmodule
```

**Fig. 2.** Fragments of a PRISM model of the Zeroconf protocol [11].

Let us consider the scenario where we have model $\mathcal{M}_1 = (S_1, \overline{s}_1, \alpha_{\mathcal{M}_1}, \delta_{\mathcal{M}_1}, L_1)$ obtained for $K = k_1$ in memory and we want to build a new model $\mathcal{M}_2 = (S_2, \overline{s}_2, \alpha_{\mathcal{M}_2}, \delta_{\mathcal{M}_2}, L_2)$ for $K = k_2$. The pseudocode for our approach can be found in Algorithm 1.

The algorithm starts with undefined constant $K$ in the *undefined_constants* variable. By executing function *get_affected_guards*, we obtain all guards that contain undefined constants. For the Zeroconf case study we obtain guards that can be found in lines 27 and 29 and for convenience we will call them $g_1, g_2$ (in the example, these commands have probability 1, but this is not a limitation of our approach).

In each of the guards, there exists a variable which is in a relation with the undefined constant $K$. Guards $g_1, g_2$ share variable *probes* with two corresponding relations: $\mathit{probes} < K$ and $\mathit{probes} = K$. We obtain such variables by calling *get_depended_variables*() and store them in variable *vars*.

The most important observation at this point is that, in order to build the model $\mathcal{M}_2$ for $K = k_2$, we do not need to re-evaluate commands in all states: it is sufficient to examine states from $\mathcal{M}_1$ that satisfied $g_1, g_2$ for $K = k_1$ but no longer satisfy for $K = k_2$, and states that now satisfy $g_1, g_2$ for $K = k_2$. To find such states we need to compute bounds on the values of *probes* for $K = k_1$ and $K = k_2$.

The task of finding bounds on a variable that is part of a guard is essentially equivalent to non-linear arithmetic involving transcendental functions which is in general undecidable [8]. Fortunately, for a large subset of practical problems, including various PRISM case studies that we have analysed, this can be accomplished using an SMT solver over the theory of linear arithmetic. In fact, for many common classes of expressions (such as this example) we can extract the bounds directly.

In our example, to compute bounds for $K = k_1$ we use function *old_min_max*() and obtain $\mathit{probes} \in [0, k_1)$ for $g_1$ and $\mathit{probes} \in [k_1, k_1]$ for $g_2$. To obtain bounds when $K = k_2$ we call *new_min_max*() and obtain $\mathit{probes} \in [0, k_2)$ and $\mathit{probes} \in [k_2, k_2]$. To find all states that satisfy guard $g_1$ for $K = k_1$ and $K = k_2$ we do a state space search using the bound $\mathit{probes} \in ([0, k_1) \cap [0, k_2))$ and store the discovered states in variable *all*. In the variable *same*, we keep states that satisfy the given guard for $K = k_1$ and still satisfy for $K = k_2$. Such states can be found by using a bound that is a conjunction on respective intervals i.e. $\mathit{probes} \in ([0, k_1) \cap [0, k_2))$ for $g_1$. All the states that need to be re-evaluated can be found in the set $\mathit{all} \setminus \mathit{same}$ which is stored in variable *reexplore*. The same process is subsequently repeated for guard $g_2$.

In the last step, we forward all states from *reexplore* to the non-incremental model construction algorithm. During the re-evalution, we may remove some transitions and make some parts of the state of $\mathcal{M}_2$ unreachable. As $\mathcal{M}_2$ must not contain any unreachable states we could use algorithms that provide dynamic reachability [18]. As our model checking algorithms will perform a decomposition of $\mathcal{M}_2$ into strongly connected components (SCCs), we use them to detect any unreachable states.

The most costly part of the algorithm is the $\mathit{find\_states}()$ function since, in the worst case, it has to traverse the whole state space to find states that satisfy a given bound. In practice, case studies often have only a small number of dependent variables and, by keeping state space ordered by a given variable and using binary search, we can significantly speed up the search process.

To conclude this section, we provide a formal proof of the correctness of Algorithm 1.

**Algorithm 1** Incremental model construction

```
1: $guards \leftarrow get\_affected\_guards(undefined\_constants)$
2: $reexplore \leftarrow \emptyset$
3: for each $g$ in $guards$ do
4:     $vars \leftarrow get\_depended\_variables(g, undefined\_constants)$
5:     for each $v$ in $vars$ do
6:         $old = find\_states(v, old\_min\_max(g, v))$
7:         $new = find\_states(v, new\_min\_max(g, v))$
8:         $same \leftarrow old \cap new$
9:         $all \leftarrow old \cup new$
10:        $reexplore \leftarrow reexplore \cup (all \setminus same)$
11:    end for
12: end for
13: $model \leftarrow model\_construction(model, reexplore)$
14: $model \leftarrow reachability(model)$
```

**Theorem 1.** *The incremental model construction algorithm constructs the same state space as the non incremental algorithm.*

*Proof.* The key idea behind the proof is to firstly prove that variable *reexplore* contains all the states from $S_1$ that may gain or lose a transition in $S_2$, i.e. for every $s \in S_1$, whenever $\delta_{\mathcal{M}_1}(s) \neq \delta_{\mathcal{M}_2}(s)$, then $s \in reexplore$. Subsequently, we have to prove that all new states in $S_2$, i.e. $s \in (S_2 \setminus S_1)$, are reachable on a path going through one of the states in *reexplore* or on a path that starts in a new initial state and consists only of states that belong to $S_2 \setminus S_1$. Lastly, we need to prove that $S_2$ does not contain any unreachable states.

**Lemma 1.** *For every $s \in S_1$, whenever $\delta_{\mathcal{M}_1}(s) \neq \delta_{\mathcal{M}_2}(s)$, then $s \in reexplore$.*

*Proof.* We give a proof by contradiction for the case when we add a transition; removing a transition can be handled in a similar way. Let us assume there exists a state $s \in S_1$ satisfying $\delta_{\mathcal{M}_1}(s) \neq \delta_{\mathcal{M}_2}(s)$ but not contained in *reexplore*. When a new transition is added in a state $s \in S_1 \cap S_2$ there must exist a guard $g'$ in $\mathcal{M}_2$ satisfying state $s$ and a syntactically equivalent guard $g$ in $\mathcal{M}_1$ that is not satisfied in state $s$. Each guard consists of boolean expressions and there exists expression $e \in g$ such that $e$ was not satisfied by $s$ but its syntactically equivalent expression $e' \in g'$ is satisfied by $s$. Bound $b$ on each variable contained in $e$ is returned by $old\_min\_max()$ and bound $b'$ on the same variable contained in $e'$ is returned by $new\_min\_max()$. As $s$ does not satisfy $b$ but satisfies $b'$ from the definition of lines $8, 9, 10$, $s$ is going to be included in *reexplore* which contradicts our initial assumption.

**Lemma 2.** *All new states in $S_2$, i.e. $s \in (S_2 \setminus S_1)$, are reachable on a path going through one of the states in reexplore, or on a path that starts in a new initial state and consist only of states that belong to $S_2 \setminus S_1$.*

*Proof.* Paths that start in a new initial state ($\overline{s}_{\mathcal{M}_2} \neq \overline{s}_{\mathcal{M}_1}$) and consists, only of states in $S_2 \setminus S_1$, are handled by the non-incremental algorithm, so the proof follows directly. For new states reachable on paths going through *reexplore* we will prove existence of such paths by contradiction. Let us assume that a new state $s \in (S_2 \setminus S_1)$ is reachable on a path that does not include any state from *reexplore*. As this path cannot contain states in $S_2 \setminus S_1$ only (as this case is covered by the first part of the proof), we move backwards on this path and find state $s \in S_1$ and its successor on a path $s' \in (S_2 \setminus S_1)$. For such state $s$ there exists an action $a \in A_{\mathcal{M}_2}(s)$ such that $\delta_{\mathcal{M}_2}(s,a)(s') > 0$. As $s' \notin S_1$ we have that $\delta_{\mathcal{M}_1}(s,a)(s') = 0$, and from Lemma 1 we know that state $s$ has to belong to *reexplore* which contradicts our assumption.

**Lemma 3.** *$S_2$ does not contain any unreachable states.*

*Proof.* Follows trivially from the definition of *reachability* function.

## 4 Incremental Quantitative Verification

Next, we consider incremental techniques for verifying MDPs. As discussed earlier, the key part of verifying quantitative properties reduces to a numerical computation phase which determines the minimum or maximum probability of reaching a set of target states, using techniques such as value iteration and policy iteration.

### 4.1 SCC-based Policy Iteration

Previous incremental verification techniques for MDPs [14] were based on the use of value iteration, applied to a decomposition of the the model into its strongly connected components (SCCs) [3]. These methods are not directly applicable to the scenarios we consider in this paper since, unlike [14], we permit structural changes to be made to the MDP. Instead, we propose an SCC-based version of policy iteration.

As in SCC-based value iteration, we first decompose the state space of the MDP into SCCs. Given an MDP $(S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, we partition $S$ into the set of SCCs $C_1, C_2, \ldots, C_m$ such that $C_i \subseteq S$ for all $1 \leq i \leq m$, $C_i \neq C_j$ for all $1 \leq i \neq j \leq m$ and $\bigcup_i C_i = S$. Then, we use policy iteration, instead of value iteration, to compute the probabilities for each SCC. Our experimental results show that SCC-based policy iteration outperforms SCC-based value iteration on certain case studies. As discussed in [14], independent SCCs can be processed in parallel to utilise advantage of multi-cores architecture in modern CPUs. In addition, the Tarjan order proposed in [15] has also been found useful for SCC-based policy iteration. For an SCC, the Tarjan order among states is the one in which states are visited by the Tarjan's algorithm during SCC decomposition. Compared to the order generated by model construction, using this order during policy iteration reduces the running time.

### 4.2 Incremental Policy Iteration

Although policy iteration can use arbitrary memoryless deterministic adversary as a starting point, a good starting adversary can reduce the number of iterations performed before policy iteration terminates. An extreme example is that the starting adversary is already optimal with respect to the reachability probability. In this case, only one iteration is needed. However, it is hard to predict an optimal adversary. A common practice is to choose the first action $a \in A_{\mathcal{M}}(s)$ if $A_{\mathcal{M}}(s)$ is ordered, or randomly choose an action in $A_{\mathcal{M}}(s)$, for each state $s$, when constructing the starting adversary.

For incremental policy iteration, we can use the results from the previous round of verification to guide the selection of the starting adversary. Let $\mathcal{M}_1 = (S_1, \overline{s}_1, \alpha_{\mathcal{M}_1}, \delta_{\mathcal{M}_1}, L_1)$ be an MDP and $\sigma_1$ an optimal adversary computed by policy iteration on $\mathcal{M}_1$. Let $\mathcal{M}_2 = (S_2, \overline{s}_2, \alpha_{\mathcal{M}_1}, \delta_{\mathcal{M}_2}, L_2)$ be the MDP that is constructed incrementally from $\mathcal{M}_1$ according to a new set of parameter values. Let $S' \subseteq S_1 \cap S_2$ be the set of states such that for all $s \in S'$:

1. $A_{\mathcal{M}_1}(s) = A_{\mathcal{M}_2}(s)$,
2. for each $a \in A_{\mathcal{M}_2}(s)$,
   (a) $\delta_{\mathcal{M}_1}(s,a)(s') = \delta_{\mathcal{M}_2}(s,a)(s')$ for each $s' \in S'$,
   (b) $\delta_{\mathcal{M}_1}(s,a)(s') = 0$ for each $s' \in S_1 \backslash S'$,
   (c) $\delta_{\mathcal{M}_2}(s,a)(s') = 0$ for each $s' \in S_2 \backslash S'$.

Intuitively, $S'$ is the set of states that have same outgoing transitions in both $\mathcal{M}_1$ and $\mathcal{M}_2$. In incremental policy iteration, we set $\sigma_2(s) = \sigma_1(s)$ for each state $s \in S'$, where $\sigma_2$ is the starting adversary when we apply policy iteration to $\mathcal{M}_2$. The reason for doing this is that the two models could have similar behaviour in those states that are affected by the change of parameter values. For states outside $S'$, we use the same strategy as that in normal policy iteration.

## 5 Experiments

We implemented our techniques in an extension of PRISM, using its explicit-state model checking engine. We evaluate our techniques on four case studies:

- *zeroconf*: a model of Zeroconf dynamic configuration protocol [11] for a device obtaining an IPv4 address from a network with $N$ existing devices by sending $K$ probes. The property being checked is "the maximum probability of the host not getting a fresh IP address in the end".
- *mer*: a model of the flight software for JPL's Mars Exploration Rover [6], where $N$ threads compete for a set of resources. The property being checked is "the maximum probability that mutual exclusion is not violated within 2 cycles of system execution".
- *consensus*: a model of the shared coin protocol with $K$ rounds of coin flips used in the randomised consensus algorithm [12] for $N$ processes. The property being checked is "the maximum probability that the protocol terminates without reaching consensus".

– *firewire*: a model of the tree identify protocol of the IEEE 1394 (FireWire) procotol [13]. The property being checked is “the maximum probability of choosing a leader within the *deadline*”.

Experiments were conducted on a PC with an 2.8GHz Xeon processor and 32GB of RAM, running Fedora 14, taking the form of running a series of verifications where a model parameter was increased each time. Tables 1 and 2 show the parameter ranges, the sizes of the resulting models, and the total time required to perform model construction and verification for all models, in both incremental and non-incremental fashion. The overhead on memory usage is negligible for both algorithms, and thus omitted.

The incremental model construction algorithm performs better than the standard PRISM model construction algorithm on each case study. The factor that has the greatest influence on the performance, is the number of states that needs to be added between each experiment. The largest number of states is added for *zeroconf* and for this case study improvements are only 2-fold. The smallest number of states added between experiments happens for *firewire* case study, but because of larger number of experiments we observe better results for the *mer* case study where improvements are almost 10-fold.

Table 2 contains results for improved model checking algorithms. For formatting reasons we used VI and PI for value iteration and policy iteration. SCC-based policy iteration performs better for each case study than the original algorithms. When compared to SCC-based value iteration, we see improvements only for the *consensus* case study. Incremental policy iteration performs better than SCC-based policy iteration in almost every case. The best results are obtained for the *zeroconf* case study where we can observe almost 2-fold speed-up. A small slowdown can be observed for one case in *zeroconf* and *firewire*. Comparing to SCC-based value iteration, incremental policy iteration in most cases performs similarly, with slowdown for *zeroconf* and 2-fold speed-up for *consensus*.

| Model | | | Time(s) | |
|---|---|---|---|---|
| Name | Parameters | States [$10^3$] | Original model construction | Incremental Model Construction |
| *zeroconf* $[N, K]$ | 10,1-5 | 32-496 | 15.9 | 12.3 |
| | 10,10-20 | 3002-5812 | 859.7 | 320.2 |
| | 60000,1-5 | 32-496 | 16.2 | 11.6 |
| | 60000,10-20 | 3002-5812 | 853.9 | 313.7 |
| *mer* $[N]$ | 1-100 | 8-592 | 429.5 | 44.3 |
| | 200-300 | 1183-1774 | 2352.4 | 192.5 |
| | 400-500 | 2364-2955 | 4375.7 | 358.3 |
| *consensus* $[N, K]$ | 2,1-40 | 1-56 | 0.8 | 0.4 |
| | 2,80-120 | 10-15 | 2.3 | 0.9 |
| | 4,1-20 | 12-20 | 15.5 | 4.8 |
| *firewire* [*deadline*] | 1000-1050 | 369-398 | 62.3 | 10.3 |
| | 2000-2050 | 970-1000 | 160.5 | 25.7 |
| | 3000-3050 | 1571-1601 | 265.7 | 42.4 |

**Table 1.** Performance comparison for incremental model construction.

| Model | | | Time(s) | | | | |
|---|---|---|---|---|---|---|---|
| Name | Parameters | States | Original VI | Original PI | SCC Based VI | SCC Based PI | Incremental PI |
| *zeroconf* $[N, K]$ | 10,1-5 | 32-496 | 56.6 | 58.1 | 8.6 | 10.6 | 8.5 |
| | 10,10-20 | 3002-5812 | 5020.2 | 5516.6 | 273.9 | 1859.1 | 1329.2 |
| | 60000,1-5 | 32-496 | 133.7 | 332.1 | 13.2 | 49.7 | 50.9 |
| | 60000,10-20 | 3002-5812 | 10242 | 16844.1 | 801.6 | 9333.9 | 4218.4 |
| *mer* $[N]$ | 1-100 | 8-592 | 82.8 | 85.4 | 65.7 | 70.5 | 65.5 |
| | 200-300 | 1183-1774 | 464.2 | 480.9 | 371.1 | 400.6 | 369.7 |
| | 400-500 | 2364-2955 | 841.7 | 885.1 | 646.5 | 695.9 | 683.3 |
| *consensus* $[N, K]$ | 2,1-40 | 1-56 | 249.6 | 160.6 | 50.1 | 33.6 | 23.2 |
| | 2,80-120 | 10-15 | 12148.5 | 6881.3 | 2286 | 1235.8 | 900.1 |
| | 4,1-20 | 12-20 | 7188.6 | 4557.4 | 1058.3 | 1029.1 | 666.5 |
| *firewire* [*deadline*] | 1000-1050 | 369-398 | 282.9 | 277.5 | 39.3 | 38.5 | 37.7 |
| | 2000-2050 | 970-1000 | 711.7 | 710.5 | 97.8 | 99.7 | 97 |
| | 3000-3050 | 1571-1601 | 1175.4 | 1160.8 | 176.4 | 174 | 181.6 |

**Table 2.** Performance comparison for incremental model checking.

## 6 Conclusions

We have described ongoing work to develop incremental verification techniques for Markov decision processes, aimed at improving the efficiency of runtime methods for systems with probabilistic behaviour. Future directions include evaluating presented techniques on a deployed adaptive system and improving system reconfiguration using policies obtained from model checking.

**Acknowledgements** The authors are part supported by ERC Advanced Grant VERIWARE, EU FP7 project CONNECT and EPSRC project EP/F001096/1.

## References

1. Calinescu, R.: When the requirements for adaptation and high integrity meet. In: Proc. 8th Workshop on Assurances for Self-Adaptive systems. pp. 1–4 (2011)
2. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimisation in service-based systems. IEEE Transactions on Software Engineering 37(3), 387–409 (2011)
3. Ciesinski, F., Baier, C., Größer, M., Klein, J.: Reduction techniques for model checking Markov decision processes. In: Proc. QEST'08. pp. 45–54. IEEE (2008)
4. Crow, J., Rushby, J., Struss, P.: Model-based reconfiguration: Diagnosis and recovery (1994)
5. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by runtime parameter adaptation. In: Proc. ICSE'09. pp. 111–121. IEEE (2009)
6. Feng, L., Kwiatkowska, M., Parker, D.: Automated learning of probabilistic assumptions for compositional reasoning. In: Proc. FASE'11. LNCS, vol. 6603, pp. 2–17. Springer (2011)

7. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proc. ICSE'11. pp. 341–350. ACM, New York, NY, USA (2011)
8. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. Journal on Satisfiability, Boolean Modeling and Computation 1, 209–236 (2007)
9. Kemeny, J., Snell, J., Knapp, A.: Denumerable Markov Chains. Springer, 2nd edn. (1976)
10. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Proc. CAV'11. LNCS, vol. 6806, pp. 585–591. Springer (2011)
11. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. FMSD 29, 33–78 (2006)
12. Kwiatkowska, M., Norman, G., Segala, R.: Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In: Proc. CAV'01. LNCS, vol. 2102, pp. 194–206. Springer (2001)
13. Kwiatkowska, M., Norman, G., Sproston, J.: Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. Formal Aspects of Computing 14(3), 295–318 (2003)
14. Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for Markov decision processes. In: Proc. DSN-PDS'11. pp. 359–370. IEEE (2011)
15. Kwiatkowska, M., Parker, D., Qu, H., Ujma, M.: On incremental quantitative verification for probabilistic systems. In: Proc. Higher-Order Workshop on Automated Runtime verification and Debugging (HOWARD-60). Springer (2011), to appear
16. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: Proc. Companion of ICSE'08. pp. 899–910. ACM (2008)
17. Parker, D.: Implementation of Symbolic Model Checking for Probabilistic Systems. Ph.D. thesis, University of Birmingham (2002)
18. Roditty, L., Zwick, U.: A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In: Proc. STOC'04. pp. 184–191. ACM (2004)
19. Wongpiromsarn, T., Ulusoy, A., Belta, C., Frazzoli, E., Rus, D.: Incremental temporal logic synthesis of control policies for robots interacting with dynamic agents. In: Proc. IROS'12 (2012), to appear
20. Woodside, C.M., Litoiu, M.: Performance model estimation and tracking using optimal filters. IEEE Trans. Softw. Eng. 34(3), 391–406 (May 2008)